

Python. Юнит-тесты. Итераторы. Исключения. Классы.

Лекция 7

Преподаватель: Дмитрий Косицин

Юнит-тесты

Doc tests. Unit tests. Библиотеки тестирования.

© Dzmitryi Kasitsyn

Fall 2017 • 3

Итераторы и генераторы

Последовательности. Итерируемые объекты. Итераторы. Генераторы.
Дополнительные способы итерирования.

© Dzmitryi Kasitsyn

Fall 2017 • 9

Обработка ошибок

Подходы к обработке ошибок. Исключения. Предупреждения.

© Dzmitryi Kasitsyn

Fall 2017 • 22

Менеджеры контекста

© Dzmitryi Kasitsyn

Fall 2017 • 33

Классы в Python

Статические методы и методы класса. Свойства.

© Dzmitryi Kasitsyn

Fall 2017 • 39

Ответы и полезные ссылки

© Dzmitryi Kasitsyn

Fall 2017 • 45

ЮНИТ-ТЕСТЫ

...

Doc tests. Unit tests. Библиотеки тестирования.

Юнит-тесты

Общая идея юнит-тестов

- Разбить код на независимые части (юниты)
- Тестировать каждую часть отдельно

Преимущества

- Нужно меньше тестов
- Проще отлаживать

Недостатки

- Нужны тесты, проверяющие взаимодействие юнитов

doctest

Библиотека **doctest** ([Py2](#), [Py3](#)) позволяет расположить тест непосредственно в документации, чтобы и показать , и проверить, как работает функция.

Недостаток подхода в его сложности: проверка некорректных входных данных или результатов сложных типов трудоемка.

```
def factorial(n):  
    """Return the factorial of n, an exact integer >= 0.  
  
    >>> factorial(5)  
    120  
    """  
    pass # implementation is here  
  
if __name__ == '__main__':  
    import doctest  
    doctest.testmod()
```

unittest

Тесты можно писать, используя библиотеку **unittest** ([Py2](#), [Py3](#)), что позволяет, в частности, группировать тесты в test cases, а также использовать множество удобных проверок.

```
import unittest
```

```
class TestFactorial(unittest.TestCase):  
    def test_simple(self):  
        self.assertEqual(factorial(5), 120)
```

```
if __name__ == '__main__':  
    unittest.main()
```

Возможности unittest

- Проверка различных типов и видов возвращаемых значений: *assertTrue*, *assertIsNone*, *assertAlmostEqual*, *assertRaisesRegex* и др.
- Возможность подготовить тестирование: методы *setup* (*setUpClass*) и *teardown* (*tearDownClass*) – вызываются перед и после каждого запуска теста (класса test case), создавая и удаляя используемые объекты.
- Возможность пропустить тест по некоторому условию (см. `unittest.SkipTest`).

Также есть библиотека **pytest**, в которой все проверки можно проводить с помощью **assert** statement'ов, и библиотека **nose**, объединяющая все виды тестов.

Подмена объектов

При помощи библиотеки **mock** (в стандартной библиотеке с Python 3.3) можно подменить любой объект, будь то поток ввода-вывода *stdout*, некоторый модуль, класс, атрибут, свойство, метод. (Пример из *test_interface.py* к лабораторным).

```
from io import StringIO

class InterfaceTestCase(unittest.TestCase):
    def setUp(self):
        self._stdout_mock = self._setup_stdout_mock()

    def _setup_stdout_mock(self):
        patcher = mock.patch('sys.stdout', new=StringIO())
        patcher.start()
        self.addCleanup(patcher.stop)
        return patcher.new
```


Итераторы и генераторы

...

Последовательности. Итерируемые объекты. Итераторы. Генераторы.
Дополнительные способы итерирования.

Sequence and iterable

Последовательность (*sequence*) – упорядоченный *индексируемый* набор объектов, например, **list**, **tuple** и **str**.

У этих объектов переопределены «магические методы» `__len__` (длина последовательности) и `__getitem__` (отвечает за индексацию).

Итерируемое (*iterable*) – упорядоченный набор объектов, элементы которого можно получать по одному.

У таких объектов реализован метод `__iter__` – возвращает итератор, который позволяет обойти итерируемый объект.

Итераторы

Итератор (`iterator`) представляет собой «поток данных» – он позволяет обойти все элементы *итерируемого* объекта, возвращая их в некоторой последовательности.

В итераторе переопределен метод `__next__` (*`next`* в Python 2), вызов которого либо возвращает следующий объект, либо бросает исключение *`StopIteration`*, если все объекты закончились.

Для явного получения итератора и взятия следующего элемента используются *built-in* методы `iter` и `next`.

```
for item in sequence:
    action(item)
```

```
def for_sequence(sequence, action):    # "for" for sequence
    i, length = 0, len(sequence)
    while i < length:
        item = sequence[i]
        action(item)
        i += 1
```

```
def for_iterable(iterable, action):    # "for" for iterator
    iterator = iter(iterable)
    try:
        while True:
            item = next(iterator)
            action(item)
    except StopIteration:
        pass
```

Итераторы

Итераторы представляют собой классы, содержащие информацию о текущем состоянии итерирования по объекту (например, индекс).

После обхода всех элементов итератор «истощается» (*exhausted*), бросая исключение *StopIteration* при каждом следующем вызове `__next__`.

Замечание. Функция *next* имеет второй параметр – значение по умолчанию, которое будет возвращено, когда итератор исчерпается.

Замечание. У функции *iter* также есть второй аргумент – значение, до получения которого будет продолжаться итерирование.

Пример реализации итератора

```
class RangeIterator(collections.Iterator):
    def __init__(self, start, stop=None, step=1):
        self._start = start if stop is not None else 0
        self._stop = stop if stop is not None else start
        self._step = step # positive only

        self._current = self._start

    def __next__(self):
        if self._current >= self._stop:
            raise StopIteration()

        result = self._current
        self._current += self._step
        return result
```

Пример использования итератора

Поскольку итераторы хранят информацию о состоянии, их можно прервать и впоследствии продолжить итерироваться. *Вопрос: что выведет следующий код?*

```
>>> odd_indices_iterator = RangeIterator(1, 10, 2)
>>>
>>> for idx in odd_indices_iterator:
>>>     if idx > 5:
>>>         break
>>>     print(idx)
>>>
>>> for idx in odd_indices_iterator:
>>>     print(idx)
```

Вопрос. Пусть в **zip** передано два итератора, один из которых истощился раньше другого. Что произойдет при продолжении работы с другим итератором?

Итерируемые и истощаемые

Последовательности *итерируемы* и *не истощаемы* (можно много раз итерироваться по ним).

Итерируемые объекты (не последовательности) могут как не истощаться (**range** в Py3/**xrange** в Py2), так и истощаться (генераторы).

Итераторы *итерируемы* (возвращают сами себя) и *истощаемы* (можно только один раз обойти).

Замечание. Зачастую в классах не реализуют отдельный класс-итератор. В таком случае метод `__iter__` возвращает *генератор*.

Замечание. В Python 2 **range** возвращает список, а **xrange** – генератор.

Пример итерируемого объекта

```
class SomeSequence(collections.Iterable):  
    def __init__(self, *items):  
        self._items = items  
  
    def __iter__(self):  
        for item in self._items:  
            yield item  
  
    def __iter__(self):  
        yield from self._items    # только в Python 3.3+  
  
    def __iter__(self):    # простой и менее гибкий вариант  
        return iter(self._items)
```

Замечание. В модуле **collections** есть и другие базовые классы, например **Sequence**. Эти классы реализуют множество полезных методов, требуя переопределить лишь несколько.

Генератор

Генератор – итератор, с которым можно взаимодействовать ([PEP-255](#)).

Каждый следующий объект возвращается с помощью выражения **yield**. Это выражение *приостанавливает* работу генератора и передает значение в вызывающую функцию. При повторном вызове исполнение продолжается с *текущей* позиции либо до следующего **yield**, либо до конца функции.

Генераторы удобно использовать, когда вся последовательность сразу не нужна, а нужно лишь по ней итерироваться.

```
>>> assert all(x % 2 for x in range(1, 10, 2))
```

Замечание. Выражения-генераторы имеют вид comprehensions с круглыми скобками. При передаче в функцию дополнительные круглые скобки не нужны.

Замечания по генераторам

Конструкция **yield from** делегирует, по сути, исполнение некоторому другому итератору (Python 3.3+, [PEP-380](#)).

В Python 3 появилась возможность у генераторов (например, **range**) узнать длину генерируемой ими последовательности (метод `__len__`) и проверить, генерируют ли они определенный элемент (метод `__contains__`).

В Python 2 ввиду реализации **xrange** не принимает числа типа **long**.

Также в Python 3 есть специальный класс – **collections.ChainMap**, который представляет собой обертку над несколькими mapping'ами.

Дополнительные способы итерирования

В стандартной библиотеке есть модуль **itertools**, в котором реализовано множество итераторов:

- **cycle** – зацикливает некоторый iterable
- **count** – бесконечный счетчик с заданным начальным значением и шагом
- **repeat** – возвращает некоторое значение заданное число раз

Также есть комбинаторные итераторы:

- **product** – итератор по декартову произведению последовательностей (по сути, генерирует кортежи, если бы был реализован вложенный *for*)
- **combinations** – итератор по упорядоченным сочетаниям элементов
- **permutations** – итератор по перестановкам переданных элементов

Дополнительные способы итерирования

- **chain** – итерируется последовательно по нескольким iterable
- **zip_longest** – аналог **zip**, только прекращает итерироваться, когда исчерпывается не первый, а последний итератор
- **takewhile/dropwhile/filterfalse/compress** – отбирает элементы последовательности в соответствии с предикатом
- **islice** – итераторный аналог **slice** (не создает списка элементов)
- **groupby** – группирует последовательные элементы
- **starmap** – аналог **map**, только распаковывает аргумент при передаче
- **tee** – создает *n* копий итератора

Замечание. В Python 2 доступны **ifilter** и **izip** – итераторные аналоги **filter** и **zip**.

Замечание. В Python 3.2 появилась функция **accumulate**, которая возвращает итератор по кумулятивному массиву.

Обработка ошибок



Подходы к обработке ошибок. Исключения. Предупреждения.

Типы ошибок

Ошибки, вообще говоря, бывают

- *синтаксические* (**SyntaxError**): переменная названа 'for', некорректный отступ
- *исключения*
 - некорректный индекс (**IndexError**)
 - деление на 0 (**ZeroDivisionError**)
 - и другие

Базовый класс для почти всех исключений – **Exception**. Однако есть *control flow* исключения: **SystemExit**, **KeyboardInterrupt**, **GeneratorExit** – с базовым классом **BaseException**.

Вопрос: для чего такое разделение?

Замечание. **Exception** в свою очередь унаследован от **BaseException** ([Py2](#), [Py3](#)).

Пример работы с исключениями

```
class MyValueError(ValueError):  
    pass  
  
def crazy_exception_processing():  
    try:  
        raise MyValueError('incorrect value')  
    except (TypeError, ValueError) as e:  
        print(e)  
        raise  
    except Exception:  
        raise Exception()  
    except:  
        pass  
    else:  
        print('no exception raised')  
    finally:  
        return -1
```


Работа с исключениями

Можно создавать собственные исключения – их следует наследовать от **Exception** либо его потомком (например, **ValueError**).

Исключение бросается с помощью выражения **raise** *<исключение>*.

Основной блок обработки исключения начинается **try** и заканчивается любым из выражений – **except**, **else** или **finally**.

В блоке **except** обрабатывается исключение определенного типа и, при необходимости, бросается либо то же, либо иное исключение.

Блок **except** можно специфицировать *одним* или *несколькими* исключениями (в скобках через запятую), а присвоить локальной переменной объект исключения можно выражением **as**.

Работа с исключениями

Для обработки всех исключений стоит указывать тип **Exception**.

Замечание. Блок **except** без указания *типа* использовать нужно *крайне редко*, иначе поток управления может быть некорректно изменен.

В случае исключения в блоке **try** интерпретатор будет последовательно подбирать подходящий блок **except**. Если ни один не подойдет или ни одного блока нет, исключение будет проброшено на уровень выше (по стеку вызовов).

Блок **else** выполнится, если в блоке **try** исключений не было.

Блок `finally`

Если исключения не было, по окончании блока **try**:

- выполняется блок **else**
- выполняется блок **finally**

Если исключение в **try** было:

- выполняется подходящий блок **except** если есть
- исключение сохраняется
- выполняется блок **finally** (в блоке **finally** исключение не доступно)
- сохраненное исключение бросается выше по стеку вызовов

Очень тонкий момент: если в блоке **finally** есть **return**, **break** или **continue**, сохраненное исключение сбрасывается. Использовать такие выражения следует только если действительно необходимо!

Обработка исключений

В случае возникновения исключения в блоке **except**, **else** или **finally**, бросается новое исключение, а старое либо присоединяется (Python 3), либо сбрасывается (Python 2).

Сохраненное исключение можно получить, вызвав **sys.exc_info()** (кроме блока **finally**). Функция вернет тройку: тип исключения, объект исключения и *traceback* – объект, хранящий информацию о стеке вызовов (обработать его можно с помощью модуля **traceback**).

У исключений есть атрибуты типа *message*, однако набор атрибутов различен для разных типов. Преобразование к строке *не гарантирует* получения полной информации о типе ошибки и сообщении.

Особенности работы с Python 2

Если во время обработки исключения его нужно передать выше по стеку вызовов или бросить новое исключение, сохранив информацию о старом, можно использовать специальный синтаксис **raise**.

```
def process_exception(exc_type):  
    try:  
        raise exc_type()  
    except ValueError:  
        # some actions here  
        exc_type, exc_instance, exc_traceback = sys.exc_info()  
        # raise other exception with original traceback  
        raise Exception, Exception(), exc_traceback  
    except Exception:  
        # some actions here  
        raise # re-raise the same exception
```

Вопрос: когда может понадобиться вариант с тремя аргументами?

Особенности работы с Python 3

В Python 3 исключение доступно так же через вызов `sys.exc_info()`.

Если во время обработки будет брошено новое исключение, оригинальное исключение будет присоединено к новому и сохранено в атрибутах `__cause__` (явно) и `__context__` (неявно), а оригинальный **traceback** в атрибуте `__traceback__`.

Бросить новое исключение, явно сообщив информацию о старом или явно указав исходный `traceback`, можно так:

```
>>> raise Exception() from original_exc
>>> raise Exception().with_traceback(original_tb)
```

Замечание. Значение `original_exc` может быть **None** – в таком случае контекст явно присоединен не будет.

ПОДХОДЫ К ОБРАБОТКЕ ОШИБОК

- Look Before You Leap (LBYL) – более общий и читаемый:

```
def get_second_LBYL(sequence):  
    if len(sequence) > 2:  
        return sequence[1]  
    else:  
        return None
```

- Easier to Ask for Forgiveness than Permission (EAFP) – не тратит время на проверку:

```
def get_second_EAFP(sequence):  
    try:  
        return sequence[1]  
    except IndexError:  
        return None
```


Предупреждения

Помимо исключений, в Python есть и предупреждения (модуль **warnings**). Они не прерывают поток выполнения программы, а лишь явно указывают на нежелательное действие.

Примеры:

- **DeprecationWarning** – сообщение об устаревшем функционале
- **RuntimeWarning** – не критичное сообщение о некорректном значении

```
>>> def deprecation(message):  
>>>     warnings.warn(message, DeprecationWarning,  
...                   stacklevel=2)
```


Менеджеры контекста

...

Менеджеры контекста

В процессе работы с файлами важно корректно работать с исключениями: файл необходимо закрыть в любом случае.

Данный синтаксис позволяет закрыть файл по выходе из блока **with**:

```
with open(file_name) as f:  
    # some actions
```

Функция **open** возвращает специальный объект – *context manager*.

Менеджер контекста последовательно *инициализирует* контекст, *входит* в него и корректно обрабатывает *выход*.

Пример менеджера контекста

```
class ContextManager(object):  
    def __init__(self):  
        print('__init__()')  
  
    def __enter__(self):  
        print('__enter__()')  
        return 'some data'  
  
    def __exit__(self, exc_type, exc_val, exc_tb):  
        print('__exit__({}, {})'.format(  
            exc_type.__name__, exc_val))  
  
with ContextManager() as c:  
    print('inside context "%s"' % c)
```

Менеджер контекста

Менеджер контекста работает следующим образом:

- создается и инициализируется (метод `__init__`)
- организуется вход в контекст (метод `__enter__`) и возвращается объект контекста (в примере с файлом – объект типа **file**)
- выполняются действия внутри контекста (внутри блока **with**)
- организуется выход из контекста с возможной обработкой исключений (метод `__exit__`)

В примере будет выведено следующее:

```
__init__()  
__enter__()  
inside context "some data"  
__exit__(None, None)
```

Менеджер контекста

Замечание. Если исключения не произошло, то параметры, передаваемые в функцию `__exit__` – тип, значение исключения и *traceback* – имеют значения **None**.

Замечание. Менеджер контекста, реализуемый функцией **open**, по выходе из контекста просто вызывает метод *close* (см. декоратор *contextlib.closing*).

Менеджеры контекста используются:

- для корректной, более простой и переносимой обработки исключений в некотором блоке кода
- Для управления ресурсами

Декоратор *contextlib.contextmanager* позволяет создать менеджер контекста из функции-генератора, что значительно упрощает синтаксис.

Менеджер контекста из генератора

```
>>> @contextlib.contextmanager
>>> def get_context():
>>>     print('__enter__()')
>>>     try:
>>>         yield 'some data'
>>>     finally:
>>>         print('__exit__()')
>>>
>>> with get_context() as c:
>>>     print('inside context "%s"' % c)
__enter__()
inside context "some data"
__exit__()
```

Классы в Python



Статические методы и методы класса. Свойства.

Пример реализации ИНКАПСУЛЯЦИИ

```
class Animal(object):  
    def __init__(self, age=0):  
        self._age = age  
  
    def get_age(self):  
        """age of animal"""  
        return self._age  
  
    def set_age(self, age):  
        assert age >= self._age  
        self._age = age  
  
    def increment_age(self):  
        self.set_age(1 + self.get_age())
```


Доступ к атрибутам

Проблема: для каждого атрибута помимо методов работы с ним нужны *getter* и *setter*, иначе атрибут можно произвольно изменять извне.

```
class Animal(object):
    def __init__(self, age=0):
        self._age = age

    @property
    def age(self):
        """age of animal"""
        return self._age

    @age.setter
    def age(self, age):
        assert age >= self._age
        self._age = age
```

СВОЙСТВА

Для доступа к атрибутам используются свойства – методы с декоратором **property** (docstring свойства получается из getter'а):

- getter – @property
- setter – @<name>.setter
- deleter – @<name>.delete

Полный синтаксис декоратора **property** имеет вид:

```
age = property(fget, fset, fdelete, doc)
```

Замечание. Создать *write-only* свойство можно только явно вызвав **property** с параметром *fget* равным **None**.

Свойства как замена функций

Для того, чтобы не хранить атрибуты, напрямую зависящие от других, можно реализовать доступ к ним с помощью свойств.

```
class PathInfo(object):  
    def __init__(self, file_path):  
        self._file_path = file_path  
  
    @property  
    def file_path(self):  
        return self._file_path  
  
    @property  
    def folder(self):  
        return os.path.dirname(self.file_path)
```

Декораторы *staticmethod* и *classmethod*

Для реализации статических методов в классах используют специальный декоратор **staticmethod**, при этом параметр *self* при вызове не передается.

```
>>> class A(object):
>>>     @staticmethod
>>>     def f(a, b):
>>>         return a + b
>>>
>>> A.f(1, 2) == A().f(1, 2)
True
```

В функцию, декорированную **classmethod**, первым параметром вместо объекта класса (*instance*) передается сам класс (параметр обычно называют *cls*).

ОТВЕТЫ И ПОЛЕЗНЫЕ ССЫЛКИ

...

Полезные ссылки

- Библиотека [mock](#) и [примеры](#)
- C3-линеаризация (цепочка поиска метода среди предков):
https://en.wikipedia.org/wiki/C3_linearization